Distributed and Parallel Programming Paradigms on the K computer and a Cluster

Jérôme Gurhem, Miwako Tsuji, Serge Petiton, Mitsuhisa Sato

Distributed and Parallel Programming Paradigms on the K computer and a Cluster

Jérôme Gurhem[1], Miwako Tsuji[2], Serge G. Petiton[1], and Mitsuhisa Sato[2]

[1] *CRISTAL and Maison de la Simulation, CNRS*
University of Lille
France
[2] *RIKEN AICS*
Kobe
Japan

Abstract

In this paper, we focus on a distributed and parallel programming paradigm for massively multi-core supercomputers. We introduce YML, a development and execution environment for parallel and distributed applications based on a graph of task components scheduled at runtime and optimized for several middlewares. Then we show why YML may be well adapted to applications running on a lot of cores. The tasks are developed with the PGAS language XMP based on directives. We use YML/XMP to implement the block-wise Gaussian elimination to solve linear systems. We also implemented it with XMP and MPI without blocks. ScaLAPACK was also used to created an non-block implementation of the resolution of a dense linear system through LU factorization. Furthermore, we run it with different amount of blocks and number of processes per task. We find out that a good compromise between the number of blocks and the number of processes per task gives interesting results. YML/XMP obtains results faster than XMP on the K computer and close to XMP, MPI and ScaLAPACK on clusters of CPUs. We conclude that parallel and distributed multi-level programming paradigms like YML/XMP may be interesting solutions for extreme scale computing.

Parallel and distributed programming paradigms Graph of task components Resolution of linear system

1 Introduction

Due to the increasing complexity and diversity of the machines architectures, most of current applications may have to be optimized to take advantage of the

upcoming supercomputers. They are more and more hierarchised and heterogeneous. Indeed, there are several levels of memory, the nodes contain several multi-core processors and the network topology is more complex too. The nodes can also be mixed with accelerators like GPU or FPGA. This results in a lot of different middlewares to use those architectures. The applications highly depend on the middlewares installed on the machines on which they will be executed. Thus, an application is usually more optimized for a middleware but the application will probably not be able to take advantage of a different middleware.

The use of the current and new machines has to be easier and more efficient. Then, we need new high level parallel programming paradigms independent from the middleware to produce parallel and distributed applications. The paradigm has to manage the data automatically and allow multiple levels of programming. For instance, the YML paradigm allows the user to develop an application based on a graph of parallel tasks where each task is a parallel and distributed application written using XMP, a PGAS language. The tasks are independent and the data moving between them are managed by YML.

In this paper, we implement the Gaussian elimination as a parallel direct block-wise methods to solve dense linear systems since this method is very suitable to task-based programming paradigms. We will summarize the distributed and parallel programming model used in YML/XMP in which we implemented the parallel block method. We use XMP since it is a PGAS language and they may be used on the exascale machines. Moreover, YML already had a backend able to manage XMP tasks through OmniRPC. OmniRPC deploy the different tasks on the supercomputer since systems to deploy tasks are not built-in yet. We compare our YML/XMP implementation with our XMP and our MPI implementation of the Gaussian elimination. We also compare it to the ScaLAPACK implementation of the LU factorization to solve a linear system. We aim to contribute to find a programming paradigm adapted to the future exascale machines. We try to evaluate the efficiency of multi-level of programming models on supercomputers as we expect it to be really efficient in the future on exascale supercomputers.

The Section 2 gives more details about the YML programming paradigm and XMP. The Section 3 explains how we implemented the block-wise Gaussian elimination using YML/XMP. The section 4 shows the results obtained on the K computer with the Gaussian elimination while comparing YML/XMP to XMP. The section 5 compares the results obtained on Poincare, a cluster of CPUs, with the Gaussian and the Gauss-Jordan eliminations using YML/XMP, XMP alone, MPI and ScaLAPACK. The Section 6 outlines related work on task programming. The last section discusses about the obtained results and introduces the future work.

2 Distributed and parallel programming model

The increasing complexity and diversity of the supercomputers architecture leads to difficulties to develop adapted applications. Actually, applications tend to be optimized for a given middleware so the applications may not have the expected performances on a machine different from the targeted one. The optimizations made to follow a middleware are often important and may not be compatible with another one so optimizing an application to several middlewares may make the application difficult to maintain or modify. Therefore, we need new programming paradigms that are easier to adapt to different middlewares. In this section we present YML, a development and execution environment for parallel and distributed applications that will help with this issue then XMP, a PGAS language used in the experiments.

2.1 YML

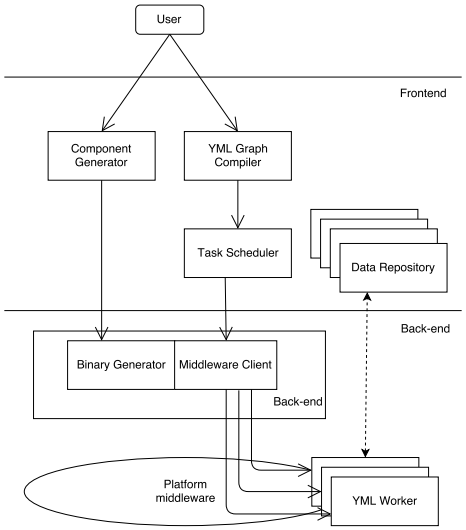
YML [1] is a development and execution environment for scientific workflow applications over various platforms, such as HPC, Cloud, P2P and Grid. YML defines an abstraction over the different middlewares, so the user can develop an application that can be executed on different middlewares without making changes related to the middleware used. YML can be adapted to different middlewares by changing its back-end. Currently, the proposed back-end [2] uses OmniRPC-MPI, a grid RPC which supports master-worker parallel programs based on multi SPMD programming paradigm. This back-end is developed for large scale clusters such as Japanese K-Computer and Poincare, the cluster of the *Maison de la Simulation*. A back-end for peer to peer networks is also available. The Figure 1 shows the internal structure of YML.

YML applications are based on a graph and a component model. They represent the front-end. YML defines components as tasks represented by nodes of a graph which is expressed using the YvetteML language. It expresses the parallelism and the dependencies between components. The application is a workflow of components execution. Components can be called several times into one application. The graph can be seen as the global algorithm of the application and the components as the core code.

The component model defines three classes of components : Graph, Abstract and Implementation. They are encapsulated using XML. They are made to hide the communication aspect and the code related to data serialization and transmission.

- Graph components contain the name, a description of the application and the graph written using YvetteML. This component also defines global parameters. They are data that can be used in components. Abstract components are called with the keyword *compute*, their name and the list of parameters they need.

Figure 1: YML software architecture



- Abstract components contain the name, the type (abstract) and a description of the component. They also define the parameters of the data coming from the global application and describe their use in the component (if the data are consumed, modified or created).

- Implementation components contain the name, the type (implementation), a description, the name of the abstract component associated, the language used, the external libraries that YML has to deploy and the source code. The links of the libraries are installed in YML then they can be used in the implementation component and YML deploys them when needed. The source code can also be written in several programming languages, including, for the moment, C/C++, Fortran or XscalableMP (XMP) [3].

These categories of component allow re-usability, portability and adaptability.

The Component generator registers the Abstract and Implementation components in YML and the data they need. The Component generator starts to check if the abstract components exist then it extracts the source code and the information needed to create a standalone application. The generator also adds the code to import and export the data from the file system. Afterwards, the Component generator calls the Binary Generator (Figure 1) to compile the generated application. It uses the compiler associated to the language filled in the Implementation component.

The Graph compiler generates the YML application by compiling the Graph component. It checks that all the components called exist, then verifies

that the YvetteML graph is correct. It also extracts the control graph and the flow graph from the Graph component and the Abstract components. It creates a binary file containing the application executable with the YML scheduler.

The scheduler manages the computational resources and the data of the application during its execution. A set of processes is allowed to YML to run the application. The scheduler explores the graph at runtime (Figure 1) to determine which component has to be run. A component can be run if the execution of the previous components is finished, the data and the computational resources are available. The scheduler runs the component of the application on a subset of the processes through a worker. The scheduler sends the component (as shown in the Figure 1) to a worker through the middleware. It executes the component on the subset of processes that manages the worker. Several workers can run at the same time if there are enough processes available.

Data are stored in a repository created in the current repository at the launch of the application. The data are read by the components that need them. The components produce or/and create data and write them in the repository.

2.2 XscalableMP

XcalableMP (XMP) [3] is a directive-based language extension for C and Fortran, which allows users to develop parallel programs for distributed memory systems easily and to tune the performance by having minimal and simple notations. XMP supports (1) typical parallelization methods based on the data-/task-parallel paradigm under the "global-view" model and (2) the co-array feature imported from Fortran 2008 for "local-view" programming.

The Omni XMP compiler translates an XMP-C or XMP-Fortran source code into a C or Fortran source code with XMP runtime library calls, which uses MPI and other communication libraries as its communication layer.

Figure 2 is an example of XMP programming. A dummy array called **template** indicates data index space and is distributed onto the nodes. Each element of the array is allocated to the node where corresponding template element is distributed.

2.3 YML/XMP

For the experiments, we use XMP to develop the YML components as introduced in [2]. This allows two levels programming. The higher level is the graph (YML) and the second level is the PGAS component (XMP). In the components, YML needs complementary information to manage the computational resources and the data at best : the number of XMP processes for a component and the distribution of the data in the processes (template). The Figure 3 shows the additional information provided to YML by the user in the implementation component in XMP. With this information, the scheduler can anticipate the resource allocation and the data movements. The scheduler creates the processes that the XMP components need to run the component. Then each process will get the piece of data which will be used in the process from the data repository.

```
#pragma xmp nodes(4)
#pragma xmp template t(0:7)
#pragma xmp distribute t(block) onto p

double a[8];
#pragma xmp align a[i] with t(i)

int main()
{
   int i;
#pragma xmp loop on t(i)
   for(i=0; i<8;i++){
     a[i]=0;
  }
}
```

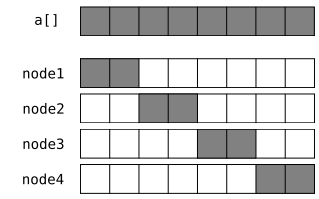


Figure 2: Example of XMP programming

Figure 3: Informations provided to YML in the implementation component in XMP

```
<?xml version="1.0"?>
<component
  type="impl" name="prodMat_xmp"
  abstract="prodMat" description="A = B x A">
<impl lang="XMP" nodes="CPU:(16,16)">
<templates>
  <template name="t" format="block,block" size="512,512"/>
</templates>
<distribute>
  <param template="t" name="B0(512,512)" align="[i][j]:(j,i)"/>
  <param template="t" name="A0(512,512)" align="[i][j]:(j,i)"/>
</distribute>
</component>
```

In this approach, there are no global communications along all the processors (like scalar products or reductions) in the application since applications are divided on component calls that use a subset of the computational resources allowed to the application. The components running at the same time don't communicate with each other and are run independently. Global communications take time and consume a lot of energy so reducing them and finding other ways to get the same results saves time and is more power efficient.

3 Parallel block implementation of the resolution of linear systems in YML/XMP

3.1 Block Gaussian method to solve linear systems

Let A be a block matrix of size $np \times np$ and $p \times p$ blocks so each block is of size $n \times n$. Let b and x be block vectors of size np and p blocks so each block is of size n. We want to solve the linear system $Ax = b$.

In the Algorithm 1, $A_{i,j}^{(k)}$ represents the k^{th} version of the block at the i^{th} row and j^{th} column of the block matrix A. Thus, $b_i^{(k)}$ represents the k^{th} version of the block at the i^{th} row of the block vector b. The algorithm is expressed using assignments on the blocks of $A_{i,j}^{(k+1)}$ and $b_i^{(k+1)}$. They consist on operations on blocks like matrix matrix product, inversion of a matrix and matrix vector product. Each assignment will correspond to a task in the implementation of the algorithms so the number of task will correspond to the number of assignments.

The Gaussian elimination [4] [5] (Algorithm 1) solves the system by doing a block triangularization of the block matrix A, and by solving the triangular system. In this algorithm, the number of assignments is $\frac{p^3+3p^2+2p}{3} \sim \frac{p^3}{3}$. We implemented the block-wise Gaussian elimination and back substitution using YML/XMP.

3.2 Block and parallel implementation using YML/XMP

For our experiments, the Algorithm 1 is expressed as YvetteML graph and the operations on blocks are implemented using XMP. Each operation is performed on a matrix of a vector which are a subdivision of the global matrices and vectors. In YvetteML, there are two means to express the parallelism between components : the *par* loop and the *par* statement. To implement the block-wise Gaussian elimination, we expressed all the call to the components in parallel and we used the event management system of YvetteML to express the dependencies between the tasks. In the Algorithm 1, we observe that each block i,j at step k of the Gaussian Elimination is updated only once. Then, it is possible to associated the corresponding tasks to the triplets (i,j,k). Each task may be launched only when some tasks of the previous steps are completed and when the data are migrated to the allocated computing resource. Therefore, the dependency graph is equivalent in this case of the precedence between triplets:

Algorithm 1: Gaussian elimination and back substitution

For k **from** *0* **to** *p-1* **do**
  (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
  (2) $b_k^{(k+1)} = Inv^{(k)} \cdot b_k^{(k)}$
  For i **from** *k+1* **to** *p-1* **do**
    (3) $A_{k,i}^{(k+1)} = Inv^{(k)} \cdot A_{k,i}^{(k)}$
  end for
  For i **from** *k+1* **to** *p-1* **do**
    (4) $b_i^{(k+1)} = b_i^{(k)} - A_{i,k}^{(k)} \cdot b_k^{(k+1)}$
    For j **from** *k+1* **to** *p-1* **do**
      (5) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k)} \cdot A_{k,j}^{(k+1)}$
    end for
  end for
end for
For k **from** *1* **to** *p-1* **do**
  For i **from** *0* **to** *p-k-1* **do**
    (6) $b_i^{(k+i+1)} = b_i^{(k+i)} - A_{i,p-k}^{(i+1)} \cdot b_{p-k}^{(p)}$
  end for
end for

for example (i,j,k) will always depend of (i,j,l<k), for the adequate value of i,j, and k. If we associate each triplet to a Yvette array of events, each assignment of the block (i,j) at step k in the Algorithm 1 have to be preceded by a wait expressing the dependence between (i,j,k) and previous triplets. For the block i of a given step of the back substitution, we use the same properties.

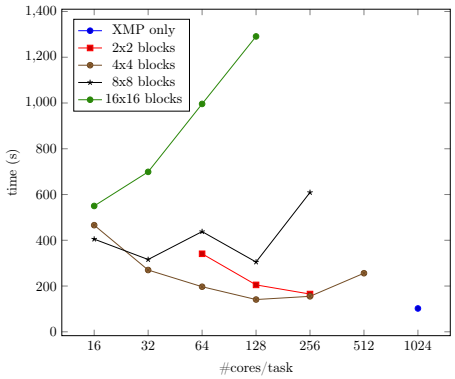
The XMP components that make the different operations are also written to take advantage of the XMP directives and use the processors allocated by YML to efficiently run the operations on the block matrices.

In the first place, we will compare the multi-level programming YML/XMP to XMP on the K computer where OmniRPC and XMP are installed and optimised. Then, we will compare YML/XMP and XMP to ScaLAPACK and our custom implementation of the resolution of a dense linear in MPI on Poincare, an IBM cluster. OmniRPC and XMP are installed on this cluster but their implementation is not optimised for this machine.

4 YML/XMP versus XMP on the K computer

This section presents the K Computer then the experiments and the results we obtained on it.

Figure 4: Resolution of linear system using Gaussian elimination + back substitution (size of 16384) on the K computer



4.1 K Computer

In this section, the tests were performed on the K Computer from Riken AICS in Kobe, Japan. This supercomputer was manufactured by Fujitsu. There is 88,128 compute nodes containing an eight-core SPARC64 VIIIfx processor and 16 Go of memory. The network is based on Tofu interconnect.

4.2 Resolution of a linear system of size 16384 on 1024 cores

This experiment consists in solving a dense linear system of size 16384 with the block-wise Gaussian elimination with YML/XMP. The matrix is already generated and each YML component has to load its data from the file system, makes the computations on the data then saves its results to the file system. The matrix is stored by blocks and each task makes an operation on blocks of the global matrix. YML expresses the operations on blocks while XMP is used to implement them. We experiment different numbers of blocks and numbers of cores per task. Only one process runs on each core. We used from 1×1 block to 16×16 blocks with tasks from 16 to 512 cores from the 1024 cores of the K Computer allocated to each run. We evaluate the time needed to solve the linear system without the generation of the matrix.

In the Figure 4, we observe the impact of the number of blocks and the number of cores per task on the execution time. We also compare the YML/XMP application to a XMP implementation of the resolution of a linear system through the non-block Gaussian elimination. We reached the best time in YML/XMP with 4×4 blocks and 128 cores per task for 141s while the XMP code took 102s.

The number of blocks determines the number of tasks in the application. For $p \times p$ blocks, there are $\frac{p^3+3p^2+2p}{3}$ tasks in the Gaussian elimination application. Thus, the application needs to have enough tasks to contain enough

parallelism without decreasing to significantly the execution time of each task. The number of blocks also influences the size of the block since the size of a block is $16384/p \times 16384/p$ values for the matrices and $16384/p$ values for the vectors in the case of a linear system of size 16384. If the number of blocks is large then the size of the block is small and the tasks on the blocks will be quick. In the opposite, a small number of blocks implies a large size of blocks so the task will be longer since it will need more operations. Moreover, if the application has too many tasks, the tasks execution will not compensate the overhead from YML.

The number of cores per task sets the number of YML workers since all the tasks use the same amount of cores and the total number of cores is fixed. Each worker launches a task at a time so there is the same number of parallel tasks as there is number of workers. For instance, we use 1024 cores in total and 128 cores per task so there are 8 parallel tasks maximum. The number of blocks needs to be high enough to use all the workers most of the time or there is unused compute power and the application will take more time. The number of cores per task also influences the speed of the execution of a task since a large number of cores will have more compute power but will also introduce more communications between cores. On the opposite, a small number of cores will induce less communications but the task will take more time.

A compromise between the number of blocks and the number of cores per task is mandatory to obtain good performances. The number of blocks gives enough tasks so there is a great number of those tasks that can be run in parallel and determines the size of the data in the task. The number of cores per task gives the number of parallel workers and the execution time of each task. The good compromise gives enough parallel tasks so the workers are busy most of the time and the execution time of the tasks is balanced with the size of the data that need to be treated. It results in applications that use efficiently the available resources.

In this case, XMP was faster than YML/XMP. We think that the number of cores is too small thus the overhead from YML doesn't compensate for direct communications across all the cores. Then, we tried to solve a bigger system on a higher number of cores.

4.3 Prediction of the optimal parameters

The prediction of the optimal values for the number of blocks (size of the block) and the number of processes per task (number of parallel tasks) is not a deterministic problem since it depends on the machine and the available resources. On a given machine, the job manager will allow the unused resources which differ each time a job is submitted. Thus, the communications time between two cores will depend on their distance. The execution time of a task also depends on the loading of the machine since each user can use the network. Moreover, the computation time has to be higher than the scheduling time so it is worth scheduling the tasks. The execution time of the application will also depend on the scheduling strategies since it will change the order of execution of the tasks.

Table 1: Execution time (s) to solve a linear system with the Gaussian elimination on the K computer with 8096 cores

Size	YML/XMP			XMP
	blocks	cores/task	best time	
32768	4×4	512	276.8	508.5
65536	8×8	512	690	2512

The parameters are highly related. Indeed, the size of a block depends on the number of blocks which influences the total number of tasks. Furthermore, the size of the block will define the number operation in the task while the number of communications will depend on the number of processes and the size of the block.

To conclude, the prediction of the optimal values is difficult but, for a given machine, the machine learning may be able to give an estimation of the value of the parameters.

4.4 Resolution of a linear system of size 32768 and 65536 on 8096 cores

We also made experiments on larger systems with more cores : 32768 and 65536 on 8096 cores. The Table 1 summarize the execution times obtained.

In this experiment, we increased the size of the system and the number of cores. In this case, YML/XMP performed better than XMP alone for the two different size whereas XMP was faster than YML/XMP on smaller cases. This is mainly due to the fact that YML doesn't make any global communications like broadcast over all the cores allocated to the application. Indeed, each task runs on a subset of the resources and the communications between tasks are made through the data server. Although, the overhead of YML is noticeable (this will be discussed in the next section), YML/XMP runs faster than XMP alone on 8096 cores of the K Computer. Thus YML/XMP and the task programming languages may be a good solution to develop and execute complex applications on huge numbers of cores on large super-computers.

We compared YML/XMP to XMP for several cases on the K computer to evaluate the multi-level distributed/parallel programming associated to the studied programming paradigm on such supercomputer, even if the system of these machines does not yet propose smart scheduling. As YML and XMP were introduced by authors of this paper, as the OmniRPC middleware used fore these experiments, we have to evaluate such programming on a more general cluster and compare it to more classic end-user programming. Then, in the next section, we compare YML/XMP and XMP to ScaLAPACK and our MPI implementation of the resolution of a dense linear system on Poincare, an IBM cluster.

5 Evaluation and analysis on a cluster

5.1 Poincare

The tests were performed on Poincare, the cluster of *La Maison de la Simulation* in France. It is an IBM cluster mainly composed of iDataPlex dx360 M4 servers, hosted at the CNRS supercomputer centre in Saclay, France. Several research teams work in this computer and we did not have dedicated access or any special assistance. There is 92 compute nodes with 2 Sandy Bridge E5-2670 processors (8 cores each, so 16 cores per nodes) and 32 Go of memory. The file system is constituted of two parts : a replicated file system with the homes of the users and a scratch file system with a faster access from the nodes. The network is based on QLogic QDR InfiniBand.

5.2 The experiments

We performed the same tests on YML/XMP as on the K Computer. We also compared those results to ScaLAPACK, to our custom MPI implementation of the non-block Gaussian elimination and 1×1 block in YML/XMP.

In the case of 1×1 block, there is only one assignment performed on a dense linear system with a XMP component. The component which runs the resolution of linear system is developed using XMP so we can evaluate the time that YML need to schedule the data, import it, run the XMP application then export the data. The data is loaded from the file system. This is the worst case for YML since it does not use the advantages of YML like scheduling and high grain parallelism.

We also compare the YML/XMP implementation of the Gaussian elimination to a XMP and a MPI implementations. Finally, we also compare to a ScaLAPACK implementation of the resolution of a dense linear system through LU factorization. In each case, we load the data from disk using MPIIO then we solve the linear system and save the result vector on disk using MPIIO.

In the MPI case, we used a cyclic distribution of the columns to keep a good load balancing in the cores. Moreover, we created a distributed cyclic array type in MPI in order to use it with MPIIO. This results in a matrix well stored by columns in the file. The cyclic distribution takes more time since it is easier to store the data without reordering them during the import/export.

5.3 Results for the block Gaussian elimination

The Figure 5 shows the execution times to solve a linear system of size 16384 and 32768 by the block Gaussian elimination and the block resolution of the resulting triangular system using YML/XMP. The best execution time for YML/XMP is obtained with 4×4 blocks and 128 cores per tasks for 51.9s for the size of 16768. The 1×1 block case with YML/XMP takes more time than the XMP only program (34.5s vs 313s).

Figure 5: Resolution of linear system using Gaussian elimination + back substitution (size of 16384 on the top and 32768 on the bottom) on Poincare

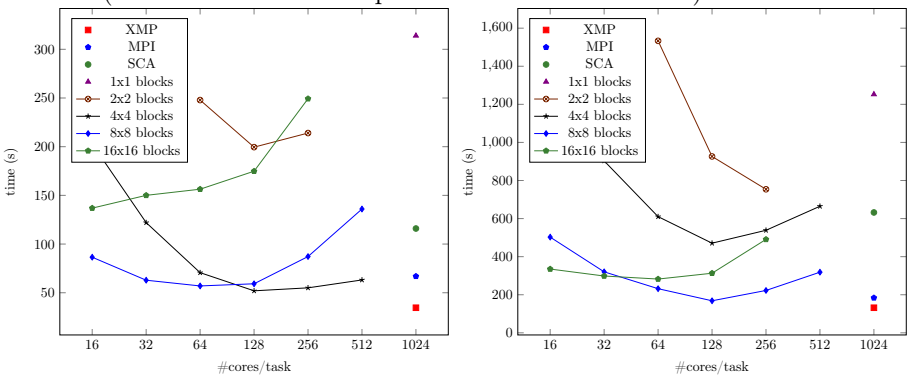


Table 2: Execution time (s) to solve a linear system with the Gaussian elimination + back substitution

Language	16384	32768
YML/XMP (best case)	51.9	168.3
YML/XMP (1×1 block)	313	1252
MPI	66.9 (60.6)	183.7 (161.9)
XMP	34.5 (32.7)	131.4 (129.6)
ScaLAPACK	115.9 (18.5)	632.1 (55.6)

The Table 2 gives a summary of the results obtained with the different programming paradigms. We have the values for two sizes of system 16384 and 32768. Moreover, for the MPI, XMP and ScaLAPACK cases, we also have the time spent in computation between parenthesis. The rest of the time is spent in IO to load and save the data as it is done for each component in YML/XMP.

5.4 Comparison between YML/XMP, XMP, MPI and ScaLAPACK

We can see a huge overhead in YML/XMP for 1×1 blocks compared to XMP alone in the Table 2. This is the case for the two sizes of system. Thus, we can expect a pretty huge overhead from YML for each component. But, the most favourable case in YML/XMP solve the linear system in 51.9s whereas XMP took 34.5, MPI took 66.9s and ScaLAPACK took 115.9s. Although, from the 66.9s only 60.6s is spent on computing. The rest is spent in MPIIO to load and save the data. In the MPI case, we used a distributed cyclic array so the data in memory and the data on disk are not mapped the same way. This is why the IO take a bit more time in MPI than XMP since, the data are stored as they are in memory without reordering them.

Moreover, we didn't optimize our MPI implementation to the maximum. There is a margin of amelioration to get faster results with our MPI implementation. This implementation is close to the algorithm and use some properties of MPI while trying to reduce the communications between the cores. In this case, we observe that YML/XMP is slightly faster than our simple MPI implementation and the XMP is relatively close to YML/XMP (It takes 1.58 times more time in the case 16384 and 1.3 in the case 32768). We can expect that with larger size of systems, YML/XMP will be closer to XMP.

Finally, ScaLAPACK provides the best results if we only consider the computation time as shown in the Table 2. Indeed, most of the time is spent in the import of the matrix. This is due to the distribution of the data in memory. In this application, ScaLAPACK uses a cyclic distribution of the matrix in a two dimensional grid of cores while the matrix is stored as huge vector representing a row-wise matrix. Then the values that have to be put in a core are scattered trough the file without any value close to another one. Thus the import of the matrix is very costly. ScaLAPACK is the fastest in the of computation time but the import of the matrix from the file system is very costly. The distribution of the data in this application using ScaLAPACK is not suited for IO.

With a good compromise between the number of blocks and the number of cores per task, we got results relatively close to XMP and MPI. Moreover, we can expect to get closer results with greater sizes of systems and even get better results at some point. YML allows more local communications than XMP and MPI. In YML, the communications are only made on a subset of cores while, in XMP, communications are made across all the cores. Furthermore, YML has a scheduler that manage the tasks and the data migrations between the tasks in order to optimize them. YML also allows to make asynchronous communications between tasks.

Figure 6: Resolution of linear system using Gauss-Jordan elimination (size of 16384) on Poincare

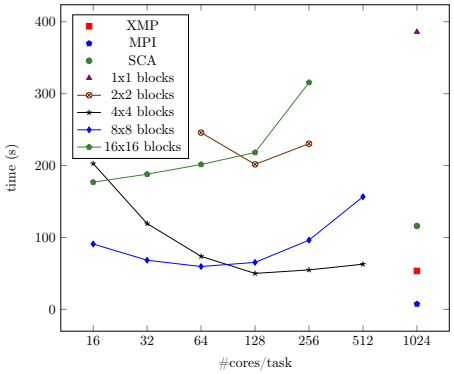


Table 3: Execution time (s) to solve a linear system with the Gauss-Jordan elimination

Language	16384	32768
YML/XMP (best case)	49.9	180
YML/XMP (1×1 block)	385	1502
MPI	7.4 (4.5)	62.1 (59)
XMP	53.4 (52.8)	289.7 (289)
ScaLAPACK	115.9 (18.5)	632.1 (55.6)

5.5 The Gauss-Jordan implementation

We also performed tests on another method to solve a linear system; the Gauss-Jordan elimination. The Figure 6 and the Table 3 show the performance we obtained on Poincare with a system of size 16384 and 32768 on 1024 cores. We performed the same experiments as for the Gaussian elimination.

In the Gauss-Jordan elimination, the operations are a bit different from the Gaussian elimination since the operations above the diagonal in the Gaussian elimination are also performed below the diagonal in the Gauss-Jordan elimination. Thus there is no resolution of triangular system. The system is solved directly. Although there is more operations, there is more parallelism between them and the critical path of the graph is shorter.

With the Gauss-Jordan method, YML/XMP runs faster than XMP on Poincare for the two sizes of systems and the difference increases with the size of the system. We observe the same effect of balance between the number of cores per task and the number of blocks as we seen previously. The parallelism inside the Gauss-Jordan elimination is greater so the task parallel programming paradigm take advantage of this parallelism to execute the application. Moreover, there is no global communications in YML/XMP but they are widely used in XMP so the increase of operations from the Gaussian elimination to the

Gauss-Jordan elimination is more impacting in XMP. Our MPI implementation is unexpectedly quite fast on a system of size 163824 but ScaLAPACK is more efficient in term of computation for the 32768 system. Finally, the Gaussian elimination performed almost two times better on XMP than the Gauss-Jordan elimination but the two methods got very close results on YML/XMP.

In this case, we globally see the same results although, the Gauss-Jordan elimination has more operations and uses more communications than the Gaussian elimination. We run faster with YML/XMP than XMP due the parallelism of the method being highly exploited by the parallel task paradigm and not using any global communications. The efficiency of the YML/XMP depends on the method implemented since the Gaussian elimination and the Gauss-Jordan elimination performed quite close on the two sizes of system while the Gauss-Jordan elimination has a significantly higher number of operations. The methods with a high parallelism are very suited to the parallel task programming paradigm and perform very well compared to XMP and ScaLAPCK. The format of the data stored in files by the data server (row-wise) wasn't suited for the distribution of the data used by ScaLAPACK so it didn't performed well as an application. The use of a better distribution of the data in ScaLAPCK and a data server making IO with the file system in YML/XMP may have shown better performances but YML/XMP and the parallel task programming paradigm show that, even with a high overhead for each task, we can get good performances.

6 Related work

The graph of tasks programming was previously used on Grid and Cloud. It has also been adapted from the distributed systems to the super-computer. Some of the main programming models using graph of tasks are listed in this section.

The DAGMan (Directed Acyclic Graph Manager) is designed to run complex sequences of long-running jobs on the Condor [6] middleware. The DAGMan language allows to describe control dependencies between tasks but there is no high level statement to easily describe a graph. Moreover, the data dependencies are not explicit so there is no optimization possible for the data communications.

Legion [7] is a data-centric parallel programming model. It aims to make the programming system aware of the structure of the data in the program. Legion provides explicit declaration of data properties and their implementation via the logical regions which are used to describe data and their locality. A Legion program executes as a tree of tasks spawning sub-tasks recursively. Each tasks specifies the logical region they will access. With the understanding of the data and their use, Legion can extract parallelism and find the data movement related to the specified data properties. Legion also provides a mapping interface to control the mapping of the tasks and the data on the processors.

TensorFlow [8] is an open source library. It aims for high performance numerical computation. A TensorFlow program is based on a Graph where each node is an Operation that process Tensors. The Tensors are multidimensional arrays used to store the data. The Operations are computations applied to the

input Tensors and may produce other Tensors. The Graph is instanced through a Session which runs the computations on the given on several available platforms like CPUs, GPUs or TPUs.

PaRSEC [9] (Parallel Runtime Scheduling and Execution Controller) is a framework which schedules and manages micro-tasks. It allows to express different types of tasks and their data dependencies. The control graph is never built and the parallel execution is only based on the data dependencies.

OmpSs [10] is based on OpenMP and StarSs. StarSs is a parallel programming model that manage tasks. Each task is a piece of code which can be executed asynchronously in parallel. OmpSs uses the Mercurium source-to-source compiler to transform the high-level directives into a parallelized version of the application and the Nanos++ Runtime Library to manage the task parallelism in the application. The tasks are expressed through the task construct via the in, out, inout clauses to express data dependencies.

CnC [11] (Current Collections) is a graph parallel programming model. A CnC graph uses three types of nodes : the step collections is a computation task, the data collections stores the data needed in the computations, and the control collections creates instances of one or more step collections. With this structure, it is possible to represent both the control and data flow graphs. In CnC, there are two ordering requirements : producer/consumer linked to data dependencies and controller/controlee linked to computation dependencies. It allows important scheduling optimizations with the two graphs directly available.

Condor and PaRSEC are middlewares created for Grid then adapted to super-computers but they don't provide all the functionalities the next generation of supercomputer will need. Argo [12] is an operating system for extreme scale computing. It allows to reconfigure the resource nodes dynamically depending on a workload, to support massive concurrency, to present a hierarchical framework for power and fault management and a communication infrastructure. Legion and OmpSs are two programming model based on data dependencies. With OmpSs, the original code is not much modified but it may not be enough to face the challenge of the exascale computing. CnC has some potential but don't seem to be used in practice since the definition of the graph is not trivial.

On the other hand, YML provides a graph programming model based on components with different back-ends which can be adapted to several middlewares. YML comes with a high-level graph language that allows the user to express parallelism and dependencies between tasks. The Abstract component gives information on the data in, out and inout of a task. With this information and the graph, YML uses the control and the data flow graph. The Implementation component contains the code of a task and can be expressed in several programming languages including C/C++, Fortran or XMP. The YML scheduler launches the tasks through the back-end. There also is a fault-tolerant version [13] of YML/XMP developed by the Japanese team which works on XMP.

7 Conclusion and future work

We experimented YML/XMP, a programming paradigm based on a graph of parallel and distributed tasks, on the cluster of the *Maison de la Simulation* in France and the K computer in Japan We solved a dense linear system of different size with 1024 and 8096 cores. We obtained results relatively close to XMP and MPI even though that YML/XMP on Poincare uses the file system to implement the asynchronous data migrations, the YML scheduler is not very efficient at this point that the overhead induced by YML is substantial. On the K computer, with a larger number of cores and and size of the system, YML/XMP ran faster than XMP. ScaLAPACK is slower because of the special distribution of the data used within it. In this application, we need to change the definition of the distribution of the data in order to reduce the import time. We can expect the same outcome on the K Computer if we run the ScaLAPACK application on it; the computation time would be excellent but the IO to import the data will slow down the application. As we are using ScaLAPACK, a YML task which make a call to the ScaLAPACK functions will not be efficient since YML load the data from disk and the distribution of the data in ScaLAPACK makes the IO difficult. Finally, we compared the Gaussian elimination to the Gauss-Jordan elimination and we find out that the Gauss-Jordan elimination has execution time close to the Gaussian elimination although the Gauss-Jordan elimination has more operations. So the parallel task programming paradigm seems well suited to execute methods with high parallelism. Thus a programming paradigm using a graph of parallel tasks where each task can also be parallel may be interesting to create efficient parallel and distributed applications on exascale super-computers.

We can outline the fact that a programming paradigm based on a graph of parallel and distributed tasks can obtain better performances on a huge amount of cores. Future supercomputers and post-petascale platforms would propose middle-ware and systems with smarter schedulers [12] and dedicated I/O systems [14] which will allow some efficient data persistence and anticipation of data migrations. In those machines, programming paradigms such as YML/XMP would be well-adapted and efficient. As the increasing number of nodes on such distributed and parallel architectures, with large latency for communications between farthest nodes, will penalize global communications, our approach which avoid global operations along all the nodes but generate operations just inside subsets of nodes, would be well-adapted.

We are planing to experiment on such machines as soon as there will be available and before we have to experiments with other applications to evaluate the assets of this paradigm.

Acknowledgements

This work is supported by the project MYX of SPPEXA, a French-Japanese-German project (ANR-JST-DFG). A part of the result were obtained by using

K computer at the RIKEN Center for Computational Science.

References

[1] O. Delannoy and S. Petiton, "A peer to peer computing framework: Design and performance evaluation of YML," in *Tird International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, I. C. S. Press, Ed., 2004, pp. 362–369.

[2] M. Tsuji, M. Sato, M. Hugues, and S. Petiton, "Multiple-spmd programming environment based on pgas and workflow toward post-petascale computing," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 480–485. DOI: 10.1109/ICPP.2013.58.

[3] J. Lee and M. Sato, "Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems," in *2010 39th International Conference on Parallel Processing Workshops*, Sep. 2010, pp. 413–420. DOI: 10.1109/ICPPW.2010.62.

[4] S. G. Petiton, "Parallelization on an mimd computer with realtime scheduler, gauss-jordan example," in *Aspects of computation on asynchronous parallel processors*, North Holland, 1989.

[5] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, "Building blocks in linear algebra," in *Numerical Linear Algebra for High-Performance Computers*, ch. 5, pp. 71–105. DOI: 10.1137/1.9780898719611.ch5.

[6] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds., John Wiley & Sons Inc., Dec. 2002.

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012, pp. 1–11. DOI: 10.1109/SC.2012.71.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. arXiv: 1603.04467. [Online]. Available: http://arxiv.org/abs/1603.04467.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013, ISSN: 1521-9615. DOI: 10.1109/MCSE.2013.98.

[10] A. Duran, E. Ayguad, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011. DOI: 10.1142/S0129626411000151.

[11] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar, "Concurrent collections," *Sci. Program.*, vol. 18, no. 3-4, pp. 203–217, Aug. 2010, ISSN: 1058-9244. DOI: 10.1155/2010/521797.

[12] D. Ellsworth, T. Patki, S. Perarnau, S. Seo, A. Amer, J. A. Zounmevo, R. Gupta, K. Yoshii, H. Hoffman, A. Malony, M. Schulz, and P. H. Beckman, "Systemwide power management with argo," in *Parallel and Distributed Processing Symposium Workshops*, IEEE, IEEE, 2016. DOI: 10.1109/IPDPSW.2016.81.

[13] M. Tsuji, S. Petiton, and M. Sato, "Fault tolerance features of a new multi-spmd programming/execution environment," in *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM '15, Austin, Texas: ACM, 2015, pp. 20–27, ISBN: 978-1-4503-3996-4. DOI: 10.1145/2832241.2832243.

[14] M. R. Hugues, M. Moretti, S. G. Petiton, and H. Calandra, "Asiods - an asynchronous and smart i/o delegation system," *Procedia Computer Science*, vol. 4, pp. 471 –478, 2011, Proceedings of the International Conference on Computational Science, ICCS 2011, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2011.04.049. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050911001074.